This is an project to be done in groups of two students and you are expected to demonstrate its operation either to
the instructor or the TA.
You are required to implement the following tree-based voting protocol for replica consistency. Let there be seven
servers, \( S_1, S_2, \ldots, S_7 \), that maintain replicas of data objects. The seven servers are logically arranged as a balanced
binary tree with \( S_1 \) being the root, and servers \( S_{2i} \) and \( S_{2i+1} \) being the left and right children, respectively, of server
\( S_i \). There are four data objects, \( D_0, D_1, D_2, \) and \( D_3 \) of type integer. Each server maintains copies of the four data
objects. Initially, all the replicas of an object are consistent, with \( D_j \) initialized to \( j \), where \( 0 \leq j \leq 3 \). There are five
clients, \( C_0, C_1, \ldots, C_4 \). All communication channels are FIFO.
The protocol for data access is as follows:

1. A client, \( C_i \), can perform the following operations on a data object \( D_j \):
   
   - Read
   - Write

   For a read operation, client \( C_i \) sends a request of the form \( \text{REQUEST}(C_i, D_j) \) to a randomly chosen server. For
   a write operation, client \( C_i \) sends a request of the form \( \text{REQUEST}(C_i, D_j, v) \) to all seven servers in the logical
tree. Having sent a request, the client starts an \text{AWAITING}_\text{GRANT} timer. The purpose of the read operation is
to retrieve the value of \( D_j \), while the purpose of the write operation is to add \( v \) to the current value of \( D_j \), where
\( v \) is an integer. The value of the \text{AWAITING}_\text{GRANT} timer is equal to 20 time units.

2. A server maintains the following \text{LOCK} information about each data object:
   
   - \text{NOT}$_\text{LOCKED}$: the object has not been locked by any client,
   - \text{READ}$_\text{LOCKED}$: a read request for this object has been granted to at least one client, and
   - \text{WRITE}$_\text{LOCKED}$: a write request for this object has been granted to client.

   Initially, state of each \text{LOCK} is set to \text{NOT}$_\text{LOCKED}$. A \text{READ}$_\text{LOCK}$_\text{COUNT} is also associated with each
\text{LOCK} and is initialized to 0. Here is how the server responds to requests from client \( C_i \) for a given data object:
   
   - If data object’s lock is in the \text{NOT}$_\text{LOCKED}$ state and a \text{READ} request is received: the request is granted,
     lock is set to \text{READ}$_\text{LOCKED}$ and \text{READ}$_\text{LOCK}$_\text{COUNT} is set to 1.
   - If data object’s lock is in the \text{NOT}$_\text{LOCKED}$ state and a \text{WRITE} request is received: the request is granted,
     and lock is set to \text{WRITE}$_\text{LOCKED}$.
   - If data object’s lock is in the \text{READ}$_\text{LOCKED}$ state and a \text{READ} request is received: the request is granted,
     lock stays set to \text{READ}$_\text{LOCKED}$ and \text{READ}$_\text{LOCK}$_\text{COUNT} is increased by 1.
   - If data object’s lock is in the \text{READ}$_\text{LOCKED}$ state and a \text{WRITE} request is received: the request is not
     granted immediately. Instead, the request is placed in a FIFO queue corresponding to the data object, lock
     stays set to \text{READ}$_\text{LOCKED}$.
   - If data object’s lock is in the \text{WRITE}$_\text{LOCKED}$ state and a \text{READ} request is received: the request is not
     granted immediately. Instead, the request is placed in a FIFO queue corresponding to the data object, lock
     stays set to \text{WRITE}$_\text{LOCKED}$.
If data object’s lock is in the WRITE_LOCKED state and a WRITE request is received: the request is not granted immediately. Instead, the request is placed in a FIFO queue corresponding to the data object, lock stays set to WRITE_LOCKED.

Whenever a READ request is granted by a server, the server sends the value of the corresponding data object in its grant message.

3. If client $C_i$’s READ request has been granted by the requested server before the expiry of its AWAITING_GRANT timer then the client does the following: (i) first $C_i$ waits for a period of time referred to as the HOLD_TIME, then (ii) client $C_i$ sends a READ_COMMIT message to the granting server, and (iii) client $C_i$ locally records the value received from the server. The HOLD_TIME is equal to 1 time unit.

4. If client $C_i$’s WRITE request has been granted by the tree of servers before the expiry of its AWAITING_GRANT timer then the client does the following: (i) first $C_i$ waits for a period of time referred to as the HOLD_TIME, then (ii) client $C_i$ sends a WRITE_COMMIT message to all servers. The HOLD_TIME is equal to 1 time unit. Granting of the WRITE request by the tree is recursively defined as:

(a) The WRITE request has been granted by the root of the tree, and either the left or the right subtree, OR
(b) The WRITE request has been granted by the left subtree and the right subtree.

If a subtree has only one server, then the granting of WRITE request by that subtree is equivalent to obtaining a grant from that server.

5. On receiving the WRITE_COMMIT message from $C_i$, all the servers perform the access operation indicated in the corresponding REQUEST message, send an acknowledgement to $C_i$. If the server was WRITE_LOCKED by $C_i$’s request then the server gets unlocked on performing the data update operation, i.e., the LOCK is set to NOT_LOCKED. Now, it can grant the request at the head of the queue of pending requests for the newly updated object.

6. On receiving the READ_COMMIT message from $C_i$ for a data object, the server decreases the corresponding READ_LOCK_COUNT by 1. If, as a result of this decrease, READ_LOCK_COUNT becomes zero, the LOCK for that data object is set to NOT_LOCKED and the server can grant the request at the head of the queue of pending requests for the object.

7. If a requesting client’s AWAITING_GRANT timer expires before it receives WRITE permission from the tree, or a READ permission for a specific server then the client withdraws its request by sending a corresponding WITHDRAW message to all servers in case of a WRITE, or to the requested server in case of a READ, and increments the number of unsuccessful read/write accesses by one. The variable to store the number of unsuccessful read accesses and unsuccessful write accesses are maintained locally at each client, and are initialized to zero.

8. On receiving a WITHDRAW message from $C_i$, servers perform the same operation as on receiving the READ_COMMIT or WRITE_COMMIT message, except for performing the data access operation.

If you believe that this protocol may result in writes to a data object being performed at different serves in different order, add safeguards to prevent such a possibility. You need to instrument your code such that if these safeguards get triggered, a corresponding message is displayed on the screen.

1 Operation

1. A client can have at most one pending access request at a time. The time between the completion (successful or unsuccessful) of the previous access and the next access attempted by a client is uniformly distributed in the range $[5,10]$ time units. Use the same distribution for the initial access. When a client wishes to perform a READ or WRITE operation it arbitrarily selects one of the five data objects for the operation and initiates the protocol as described above. To determine the type of operation, the client randomly generates a number in the range 1 through 10 (inclusive). If the number is equal to 1, the requested operation is a WRITE operation. Otherwise, the requested operation is a READ operation.
2. In your experiments all communication should be performed using IP stream sockets.

3. Execute your experiment until a total of 50 updates have been attempted.

4. For the experiment report the following:
   
   (a) For every data object, do all replicas of the object go through exactly the same sequence of updates?
   (b) The number of successful and unsuccessful READ and WRITE accesses by each client.
   (c) The total number of messages exchanged.
   (d) For the successful READ accesses, the minimum, maximum, and average time between issuing a READ request and receiving permission from the requested server.
   (e) For the successful WRITE accesses, the minimum, maximum, and average time between issuing a WRITE request and receiving permission from the server tree.

5. Repeat the experiment with the HOLD_TIME set to 0.1, 0.5, 1.5, 2.0, and 5.0 time units.

6. What is the impact, if any, of the value of HOLD_TIME on the performance of the protocol?

2 Point Distribution

Design Document (20%): Employ appropriate software engineering practices to prepare this document.

Contributions Document (5%): Specify which team member played what role (designer, coder, debugger/tester) in the implementation of various modules of the project and the amount of time spent doing so.

Implementation (30%): Source code of your well structured and well documented program. You are required to write your code in C, C++ or Java.

Correctness (35%): Output that your program produces and the statistical analysis of the results.

Report (10%): Results of all your experiments presented in an understandable form.

3 Submission Information

Submission instructions will be posted on the course discussion forum on eLearning.